THE AUSTRALIAN NATIONAL UNIVERSITY

TR-CS-00-02

A Survey of Simulation Tools for CAP Project Phase III

Peter Strazdins

October 2000

Joint Computer Science Technical Report Series

Department of Computer Science
Faculty of Engineering and Information Technology

Computer Sciences Laboratory
Research School of Information Sciences and Engineering

This technical report series is published jointly by the Department of Computer Science, Faculty of Engineering and Information Technology, and the Computer Sciences Laboratory, Research School of Information Sciences and Engineering, The Australian National University.

Please direct correspondence regarding this series to:

Technical Reports
Department of Computer Science
Faculty of Engineering and Information Technology
The Australian National University
Canberra ACT 0200
Australia

or send email to:

`Technical.Reports@cs.anu.edu.au`

A list of technical reports, including some abstracts and copies of some full reports may be found at:

`http://cs.anu.edu.au/techreports/`

Recent reports in this series:

TR-CS-00-01 Jens Gustedt, Ole A. Maehle, and Jan Arne Telle. *Java programs do not have bounded treewidth.* February 2000.

TR-CS-99-02 Samuel Taylor. *A distributed visualisation tool for digital terrain models.* July 1999.

TR-CS-99-01 Peter E. Strazdins. *A dense complex symmetric indefinite solver for the Fujitsu AP3000.* May 1999.

TR-CS-98-14 Michael Stewart. *A completely rank revealing quotient* urv *decomposition.* December 1998.

TR-CS-98-13 Michael Stewart. *Finding near rank deficiency in matrix products.* December 1998.

TR-CS-98-12 Vadim Olshevsky and Michael Stewart. *Stable factorization of Hankel and Hankel-like matrices.* December 1998.

A Survey of Simulation Tools for CAP Project Phase III

Peter Strazdins

July 2000

Abstract

This paper gives a survey of architectural simulation tools and methodologies and an evaluation of tools with respect to Theme 1 of the ANU-Fujitsu CAP Project Phase III. It thus provides background work for this Theme.

1 Introduction

Architectural performance analysis is an increasingly important technique in modern computer systems design [1]. Its main component is called *simulation*, where a model of the system is made; usually this model can reproduce the functional and (an approximation of) the intended timing behaviour of the system.

Functional simulators (also called *emulators*) are often used to generate an *instruction traces*, the sequence of instructions executed on the target system when an given application program with a given input would be run on that system. The trace can then be passed to a tool modelling the performance characteristics for performance evaluation; this is called *trace-driven simulation* (see Section 1.1).

However, simulators can also directly have such a model built into them, obviating the need for a trace; this is called *execution-driven simulation*. Generally, such an approach is more expensive but more accurate, especially for multiprocessor simulation. This is because on these systems, the relative timing of events (on different CPUs) can affect the trace itself (eg. acquiring a spinlock). Also, in the case of the UltraSPARC, memory system timing issues can affect the behaviour of code using for example block load/store instructions. Execution-driven simulation also has the advantage that the full information of the workload is available, including object file symbol tables; this allows easier mapping of low-level hardware-oriented events into higher level user-oriented events. It is also less intrusive than many trace-driven methods.

Analytical performance models are one (complementary) alternative to simulation; however, such techniques still need simulation for calibration and validation, and so are not a substitute [23]. For calibration, typically an application under a workload must be simulated (trace-driven) to obtain model parameters; for memory behaviour, these might include average times between memory requests, the probability of a write request causing a cache writeback etc [22]. This may result in speed up of ≈ 100 over execution-driven simulation [22].

In general trace-driven simulation is regarded as a reasonable compromise between speed and accuracy [8]; this is especially the case if it uses an instrumented binary (as opposed to interpreting the binary). However, memory effects of speculative execution may also be lost from the trace [1]. Methods to speed up simulation include sampling [1, 20], various compression techniques (in the case of tracing) [6], and parallelism [6].

In general with simulation there is a speed–accuracy/detail tradeoff; this presents real challenges for large workloads. One solution, akin to the idea of sampling, is enable the simulator to operate in various modes along this tradeoff, as only parts of a large simulation may require a high degree detail. This is especially critical for simulating OS effects, where for example, phenomenon such as memory fragmentation and file system caching effects change with prolonged use [9].

Architectural simulation tools are extremely large, complex systems, with intimate interaction with the (host and target) operating system and architecture. Many such tools were produced in the last 10 years, requiring between 5 and 20 person-years of effort.

However their benefits are many over the alternatives. A full machine simulator can be used for (1) analysing workloads, with full visibility into machine behaviour, (ie. collecting statistics on events that are unavailable for collection on real hardware), (2) OS development (including debugging: determinism can be ensured; also for performance tuning: it is a better alternative to modifying the kernel to add instrumentation code), (3) perform simulation before the hardware exists, and (4) can vary machine parameters for architectural studies.

Note that user-level simulators can only partially satisfy these requirements. These merely emulate systems calls, and can give little insight into the machine activity required, nor even into the resulting interference

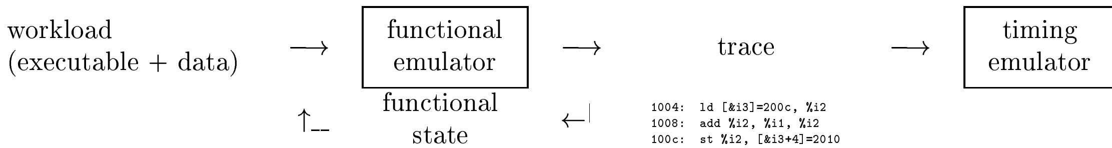

Figure 1: Trace-driven simulation

	user-level only:	**full machine**:
functional state:	(user-level) registers	all registers
	a single VM space image	multiple VM spaces (including kernel images)
emulate:	system calls	devices (boot PROM, DMA, disk, network, ...)
		MMU / TLB and physical memory (PM), bus
		(anything that can cause interrupts / exceptions)

Table 1: Comparison of requirements for user-level and full machine simulators

(eg. cache and TLB pollution) of the operating system activity [9]. DMA requests to devices, which typically occur within the kernel, cannot be captured by a user-level simulator. Generally, they can only simulate a single user process (or perhaps, by emulating thread creation, multiple threads within a process). Furthermore, user-level simulators cannot also capture behaviour involving physical addresses, including accurate analysis of) of physically indexed caches, TLB miss and page fault analysis.

1.1 Simulation Concepts and Challenges

A simplified process of trace-driven simulation is depicted in Figure 1. Note that the trace can be sampled or parts of it skipped, before passing to the timing emulator, allowing significant improvements in simulation speed.

Execution-driven simulation can then be envisaged as the functional and timing emulators being combined. The simulator generally runs as a single user-level process on a *host machine*; it interprets each instruction, updates the functional and timing *states*, possibly collecting statistics as it goes.

Table 1 gives a comparison between the two classes of execution-driven simulators. Thus, extra challenges for full machine simulators include: efficient dynamic address translation (ie. model the MMU), having sufficient speed to simulate the booting process ($\approx 10^9$ instructions on a modern OS!) and to be able to simulate arbitrary, (preferably) un-modified executables and kernel boot images, including 'badly-behaved' processes, self-modifying code, etc.

In full machine simulation, the hardware components are modelled in a modular fashion; however the *interactions* between them must also be captured. For example, the cache interacts between the CPU pipeline and the memory system. The efficient implementation of such interactions is thus non-trivial.

For SMP simulation, the cache coherency protocols must be modelled faithfully, and yet efficiently. Note that in principle, each memory access on any CPU in principle requires a synchronization of the simulated memory system, raising significant efficiency problems if realistic 'interleavings' of CPU activity is desired.

The classical model of a simulator is to *interpret* the instructions in a workload. A significantly faster technique, known as *dynamic binary translation* [4, 26, 2], translates each basic block of target code into instrumented host code. Here, to perform the functional simulation, the simulated program's state (registers + memory) is represented by an area in the host memory. The current simulated instruction is decoded and *dynamically translated* into a sequence of host instructions that will emulate this.

Figure 2 illustrates this process, for a simulator having a `dispatch_loop` which repeatedly translates the basic block at `SimPC*` and jumps to the translated code. For full-machine simulators, the places to insert code to perform dynamic address translation is marked with a '*'. Code to collect events for (run-time definable) statistics calculation can be inserted at the end of the translated block.

A *translation cache* can be used to amortize translation costs; entries in this cache must be invalidated whenever their page has been written to (eg. self-modifying code). The translated blocks can be *chained* for further performance enhancement.

Full machine simulators must also efficiently translate virtual to physical address references (and also get corresponding address in the functional state, ie. host memory). Details applying to specific solutions can be

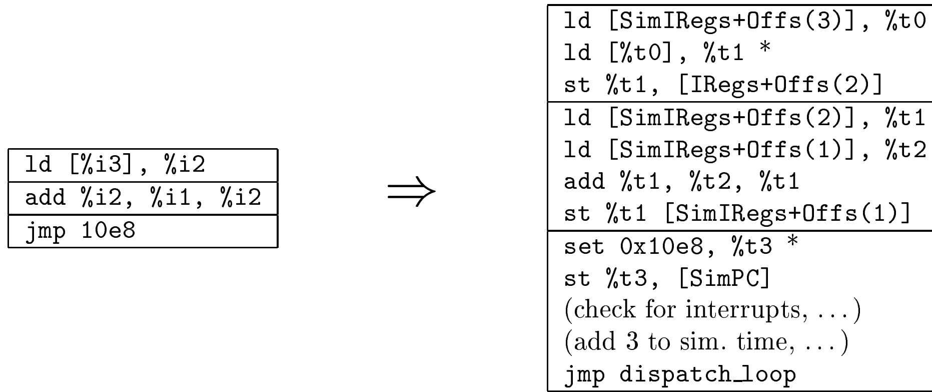

Figure 2: Dynamic Binary translation of SPARC code

found in Sections 3.2 and 3.3.4.

SMP simulation presents further challenges. The simplest technique is to simulate s cycles of each CPU in a 'round-robin" fashion; here there is a speed-accuracy tradeoff: a low value of s gives a more realistic interleaving of CPU activity but incurs more switching overhead.

An alternate solution is to have a different simulation process / thread per CPU (sharing the same simulated memory state). Thus SMP parallelism can in principle be exploited. However, as realistic interleavings require frequent synchronizations (indeed upon every possible simulated memory access, ie. upon every simulated cycle!), it remains an open problem getting an efficient and accurate SMP parallelization of am SMP simulator.

Physical memory must also be modelled upon a per-line basis, with simulator data structures organized so that it can be quickly determined which CPUs are currently accessing the line.

The simulation of distributed memory parallelism can also use a separate simulation process per node. Here, the simulator must also model the devices for message passing (eg. the network); the simulated message arrival must then cause an interrupt on the corresponding host CPU. In general, efficient parallelization on either an SMP or Distributed Memory host is much easier, due to the less frequent occurrence of messages in a DM machine (than memory operations in an SMP machine). However, accurately modelling events such as network collisions requires a finer degree of synchronization; furthermore, the need for a centralized notion of time is a problem for modelling large-scale systems [9](p98).

Finally, a simulator must be able to provide useful information on the behaviour of the workload, preferably in a high-level way which relates to the source code. For full machine simulators, working in many places upon physical addresses and multiple address contexts, this is in principle more difficult. Profiling information can include cache misses in the most heavily executed procedures, 'hot spots' in data structures, for both application and kernel code.

1.2 Requirements for Theme 1

The Themes for ANU-Fujitsu CAP Project Phase III are currently given as follows:

(1) Tools for analysing cache and memory behaviour of SMP computers

  - Development of the tools which trace memory behaviour including user applications and the operating system.
  - Investigate techniques for representing the possible interaction between threads or processes in the trace.

(2) Evaluation of the cache and memory behaviour of important applications on SMP machine

(3) Optimization of operating system for SMP machine, particularly in memory behaviour

(4) SMP and NUMA architecture

Theme 1 is in some sense the foundation, although some work in Theme 3 could proceed independently. However, it is envisioned that the tools from Theme 1 and applications from Theme 2 would be used to evaluate the effectiveness of the Optimizations of Theme 3.

The requirement of Theme 1 is to produce a simulation tool that:

1. capture kernel level activity (as well as user level activity).

   Ideally, it should be able to boot a commercial OS (eg. Linux or Solaris) with little or no modification.

2. capture thread / process interactions

3. produce traces

4. be used for an SMP simulator detailed at the cache/memory level

5. simulate/trace applications from compiled SPARC V9 binaries

6. provide debugging facilities for an OS kernel

7. be fast enough to simulate large workloads (eg. OS boot process, OLTP benchmarks etc)

Desirable attributes are high efficiency (eg. parallelism if executed on an SMP host), and being free of licensing agreements that are expensive or have source code restrictions.

As well as support the other Themes of the project, the simulator is also required to analyses the code produced by an experimental parallelizing compiler (can assume mainly user-level workloads, the number of threads = the number of CPUs) and a parallel Java garbage collection application (having both user and kernel-level locking on shared data structures, and the number of threads > the number of CPUs). SPARC V9 under Solaris would be the target platform [16].

1.3 Simulation at the FL Systems Laboratory Group: current practices

This section gives an overview of the current practices in tracing by Systems Laboratory Group; this provides the motivation for the requirements of Theme 1.

The Systems Laboratory Group have been for some time using simulation tools to guide development of (SMP) architecture, and operating systems and compilers for these machines. Their requirement is for fast simulation of large-scale (≤ 64) CPU systems.

A tracing tool called *Shade* [4] (see Section 2.1) is used to trace a (SPARC V8) application binary under a desired workload. The trace evidently includes all instructions, eg:

address	opcode	operand
`0000100`	`ld`	`00200000, [%i1]`
`0000104`	`add`	`%i2, %i1, %i1`
`0000108`	`st`	`[%i2],00201000`

It appears that all instructions appear in the trace; thus *load-use dependencies* can be estimated.

From the instruction trace generated, an FL-developed tool called *ParaTool* [10] simulates the processor and cache and generates a memory access 'sub-trace'. Evidently, some kind of cycle counting (pipeline simulation) is employed, so that the times of the memory access, and their dependent instructions, can be calculated.

The memory access sub-trace is then fed into a second FL-developed tool called *MUSCAT*, which simulates the bus model used.

Shade's limitations for this purpose are:

1. it emulates all system-level activity; thus it cannot capture such activity in its trace

2. it can only simulate a single process / thread.

The second limitation is overcome by FL by breaking loops to be parallelized into 'sections', each to be executed on a different CPU. Between these loops, special macros (expanding to `nops`) are inserted [16]. Thus only *fork-join* parallelism (possibly with barriers) can be simulated [10]. These appear in the trace, and then are interpreted by *ParaTool* as 'markers' to separate the original trace into the sub-traces that would have been executed on each CPU.

This requires modification of the source code by hand for each application to be benchmark. Because of the effort required, and the fact that the source code may not always be available, this approach is undesirable.

Furthermore, SMP related operations such as spinlocks will not appear in such a trace, nor does system level code (eg. process scheduling and synchronization, which may be significant for some applications on large scale SMP).

2 Overview of Existing Tracing Tools

This and the following section describes existing simulation tools, particularly with respect to properties relevant to the requirements of Section 1.2.

Tracing tools, while not adequate for the requirements of Section 1.2 perform the *functional simulation* common to the execution-driven tools. Thus, many of the relevant issues are encountered for these tracers.

As Shade is used by FL, it will be treated in some length.

2.1 Shade

Shade [3, 4] is an address tracer for SPARC binaries[1].

However, as well as storing traces to a file for later analysis, Shade can also feed its trace via memory to a user-supplied *analyser* (eg. ParaTool, see Section 1.3).

Shade performs *dynamic binary translation*, as discussed in Section 1.1. Note that for the SPARC, stack references have to be identified and treated somewhat differently to others[2].

Tracing instructions are also added; this involves a call to a tracing procedure (may be user defined). No timing information is recorded in the trace. In the main simulation loop, execution then jumps (via a register holding the address of this block) to this block of instructions.

Shade is flexible in the format of the information appearing in the trace, adjustable for each opcode [3]. This can be varied dynamically. The traces are buffered in memory to reduce I/O costs.

This process is sped up by means of a *translation cache*; some other optimizations are performed. This is one of the standard techniques used for simulation of binary codes.

Most system calls in the application are simulated; some can be directly executed,[3] eg. a `fork()` is executed and creates 2 instances of a Shade process. `brk()` and `sbrk()` calls are simulated by allocating more space in the application's data area. The application's file descriptors are translated into actual descriptors before use; this avoids conflicts between simulator (+ analyser) and application's descriptors (eg. standard input). Otherwise, most process-oriented information (eg. process id, `cwd`, resource usage) are shared by the simulator and the application.

A further issue is the handling of signals, which are first handled by the simulator. While Shade itself does not reserve any signals, conflicts can occur for signals for the application and the analyser. While some signals (eg. external interrupts, timers) may be 'shared' (in the sense that the analyser handling them corresponds to the application doing so), Shade must detect and subvert any signal-related system calls in the application that could interfere with the analyser's signal handling. For handling of (asynchronous) signals meant for the application, the host's signal handler queues the signal for checking in the simulator main loop. The handling of synchronous signals (eg. divide by 0 trap) generated from the application usually occurs naturally in the translated code, since here the source and target architecture are similar.

Various optimizations have been performed in Shade [3]. For example, *translation chaining* is used to translate a group of 2 or more instructions (within a basic block) together, amortising the overheads of saving simulation state and returning to the simulation loop. Note that transferring the application's condition codes to the host is expensive (this applies particularly to SPARC V8 hosts), due to SPARC having no instructions to transfer condition codes to/from registers, and Shade makes efforts to reduce this where possible.

Only user-level code on a single CPU can be traced by Shade. Particular complications with simulating kernel mode include a more complex address translation (for multiple address spaces) and memory protection checking [3]. The latter also presents a problem for extension to SMP where portions of the address space can overlap (stacks of threads) [15].

2.2 MPtrace

MPtrace [6] precedes Shade but in some sense it has more advanced features. It is designed for efficient multiprocessor tracing, using advanced optimization techniques. The host and target architecture must thus be the same.

[1] Some extensions to other target platforms has been also performed. Of primary interest, SPARC V8 (or 32-bit V9) binaries can be simulated on a SPARC V8 host

[2] Including simulating register window overflow/underflow traps. Note that for this purpose, Shade must simulate the register windows as well.

[3] System call parameters and return values corresponding to addresses must be transformed appropriately.

MPtrace requires assembler source for the application [4], taking advantage of the symbolic address to insert tracing instructions. Naturally, kernel level code cannot be traced.

MPtrace takes radical steps to minimize the *time dilation* introduced by tracing, and thus reduce the distortion due to multiprocessing effects. Firstly, intra- and inter- basic block analysis is used to aggressively optimize the inserted code. Secondly, to minimize trace size, a *roadmap* of the application is produced, which contains all statically deducible information. The roadmap is also used to remove the dilation introduced by inserting the trace instructions; these both occur on a post-processing stage.

For multiprocessor simulation, the same number of application threads are generated as on the original application.

Extra threads are used to write out traces; to preserve trace accuracy from this point, all application threads are suspended while this occurs. If the host allows threads to migrate, the OS scheduler must modified to drop a timestamp and processor number into the trace buffer.

3 Overview of Execution Driven Simulation Tools

3.1 MINT

MINT [24] is similar to Shade in many ways, except that it operates on MIPS (statically linked) binaries and supports multiprocessor simulation, albeit on a single host CPU. As for Shade, the (target) Unix system calls are directly emulated.

One difference is that, rather than generating traces per se, the simulator generates 'events' (eg. read / write) which are supplied to the (user-supplied) analyser. For this reason, it is classified as as an *execution-driven* tool.

For multiprocessor simulation, `fork()` is emulated by thread creation; this is due to lower synchronization overhead for threads. This is turn creates problems with the child thread now sharing data with the parent (this is solved by dynamically mapping the child's addresses).

When simulating threaded applications, all inter-thread communication (eg. barriers, locks, normal and shared memory allocation) must be through special library calls, which are separately emulated. This restricts the applications that can be simulated by MINT.

3.2 SimICS

SimICS [14, 15, 25] is a full system level SPARC V8 (also M8810) simulator, supporting user-defined analysers (for instruction / data cache models). It supports multiprocessor simulation (using *timeslicing*, on one host CPU), for both shared and/or distributed memory configurations. (Physical) memory profiling is possible. SimICS also can be used as an operating systems debugger (using a modified `gdb` interface).

It also supports emulation of the SunOS 5.x / Solaris 2.x ABI, allowing faster user-level simulations.

SimICS supports multiple address spaces. It can thus profile cache and TLB misses, and these can be related to source code lines. It is designed in order to make MMU and cache simulation efficient, and it thus claimed to be faster than (SimOS) Mipsy in this point [14].

It can boot un-modified Linux 2.0.30 and Solaris 2.6. This is because it has binary-compatible device simulators for devices, including SCSI, console, interrupt, timers, disks, EPROM and the Ethernet. This allows it to act as a 'virtual console'. With the exception of the EPROM, these device simulators are faithful, thus allowing I/O performance effects to be included in the simulator.

SimICS boots the OS from a fake PROM device, as this stage is not of interest to simulation. The SCSI simulator required 4500 lines of C code. Disk contents are modelled by taking dumps of real partitions. Inside SimICS, a session may thus involve a boot, log in and run an application. The Ethernet device is mapped into the physical memory of the simulated machine. The simulated disk state can be saved, allowing the simulator sessions themselves to update the disk.

SimICS uses dynamic binary instruction translation with a translation cache, similar to Shade. However, the translation is into an intermediate code, which is then interpreted. However, upon the simulation of each instruction, a code segment called a *service routine* is executed, which can update any relevant event queues etc.

The *core interpreter* of SimICS is implemented using a meta-tool called SimGEN, which produces the service routines from a high-level specification of the instruction set. These routines are typically 10–30 host

[4] While disassembling a binary is possible, the problem arises that it is hard to distinguish memory used for code and data.

instructions. SimGEN produces routines of better performance than is practical to do manually, and greatly simplifies the specification of the target instruction set [14].

For each of accuracy level of simulation available (functional, minimal statistics, memory profiling and cache simulation), different *service routines* are provided; this allows a user-definable accuracy–speed tradeoff in SimICS [15].

SimICS does not model timing at CPU pipeline level, but can supply sub-traces augmented with hardware events such as cache and TLB misses (indicating the target's *non-volatile* state), from which it is however possible to use a post-analyser to perform cycle-accurate simulation [25].

SimICS's internal structure models a hierarchy of data objects corresponding to nodes, CPUs (with shared caches), memory and devices (see Figure 1 [15]). There are corresponding C modules to manipulate these; in particular, the `local-processor.c` module implements most target-architecture dependent attributes.

The Simulator Translation Cache (STC) is used to efficiently simulate memory accesses. It allows an execution profile (on physical memory) to be kept, which is useful for performance analysis of the simulated application. The addresses can be traced via symbol tables to the source code (if available).

The STC is the key to efficient memory simulation in SimICS. For each reference to a virtual address (VA), not only the corresponding (target) physical address must be calculated, but the corresponding host virtual address (RA) also, in order to access its contents. Upon the 1st access to a virtual page, the VA will miss in the STC. Upon a miss, the MMU device will be invoked to put the corresponding page into the STC; if this implies a simulated TLB replacement, the MMU will also request any corresponding entries to be flushed from the STC.

6 separate STC are used for R, W, and E accesses for each of user and supervisor accesses; a hit to the appropriate STC implies protection access is granted. The STC is also designed to ensure a miss upon misaligned memory accesses.

To simulate a (top-level) data cache, the STC is indexed by cache lines, instead of pages (naturally the range of address that will hit the STC will be correspondingly diminished), so that an STC hit implies both a simulated TLB and cache hit.

Thus, the STC is can be used (and is efficient) in the situations where a memory access follow an optimistic path, ie. one that does not change the memory hierarchy state, which generally cache and TLB hits do not. However, if replacement schemes such as LRU are used, where the cache / TLB state are still affected by hits, this causes a problem.

A point not noted in [15] is that stores to a copy-back cache can cause the cache state to change, even upon a hit (the dirty bit get set). Presumably, it would be possible to organize the STC so that upon the first store to a clean line, the lookup will fail. Alternatively, if the top-level cache *writes through* to a second level of cache, the state of the latter would also be changed.

Separate STC and cache objects are kept for each CPU; thus timeslicing between CPUs involves merely changing the pointers to these objects.

Physical memory is modelled in the simulator on an on-demand page basis. As the STC returns only the RA upon an STC hit, it appears that the physical memory state can only be updated upon cache or paging events. Each line in a physical page has a link to the lines in (shared) cache(s) that might be holding it. This enables the implementation of cache coherency protocols.

SimICS can be dynamically extended to include new devices. Devices are assumed to be *memory-mapped*: once the STC is notified of the mapping, it redirects accesses to the corresponding (simulated) physical memory to the device. The other extension is for new memory hierarchy modules.

The back-end of SimICS can be user supplied, like Shade, and can include debuggers. As the interface can handle multiple address contexts, the user can debug several user programs (and the OS) at one time [14].

3.3 SimOS

SimOS [21, 20, 26] is a complete machine simulation system containing several simulator *cores*. These cores use a common state representing the machine and simulated workload; thus it is possible to dynamically switch between them. This enables realistic workloads to be studied at a desired level of detail, with the 'uninteresting' sections of the workload 'fast-forwarded' over by a faster core.

These cores differ in the speed–simulation detail (accuracy) tradeoff: thus a fast core, eg. Embra, can be used to *position* the workload (eg. simulate the boot process), or to skip quickly through uninteresting sections. A more detailed core, eg. Mipsy, can then simulate the desired parts of the workload with greater detail. Statistical sampling techniques, analogous to those used for trace-driven simulators, can be used to determine when to

switch between cores, and to evaluate how representative are the parts of the workload that were simulated by the more detailed cores [20].

Like SimICS, SimOS also has software modules to emulate the various hardware devices; these also incorporate timing models to some degree. See Figure 2.1 of [26] for a schematic describing the SimOS environment.

Embra has interesting mechanisms to obtain its speed; it will be described in detail in Section 3.3.4.

Mipsy can be used to accurately model a memory system with two levels of cache. It is typically an order of magnitude slower then Embra [26]. Further details are given in Section 3.3.5.

A third core, called MXS, can accurately model a post-RISC CPU in great detail, including pipeline stalls, branch prediction and speculative execution. It is typically an order of magnitude slower then Mipsy.

SimOS has been used to study large workloads [20] and also for the development of the Hive Operating System [26]. Like SimICS, it can be interfaced to debuggers. SimOS can also provide *checkpoints*, where the simulator state can be saved and restarted from any desired point: this is useful for example for skipping over the 10's of billions of instructions required to setup a workload (from the boot stage). Furthermore, a single checkpoint can be restored between multiple hardware configurations for architectural studies [9].

As well as simulating SMP machines, it appears that SimOS can model multiple machines connected by a (LAN) network [9](p100).

Currently, SimOS has been ported to MIPS 3000 / IRIX 5.3 (© 1996) and Alpha / DEC Unix (© 1998). Its source code is available, and may be modified for non-commercial purposes.

3.3.1 Device Simulation

A full machine simulator must provide the same interfaces for the hardware that the workload running on the target host (and OS) expects. The minimal devices include a disk (filesystem containing OS and application files), a timer, and a console.

Many devices are I/O mapped; thus the memory sub-system must simulate memory-mapped or DMA accesses expected by all simulated devices [9]. SimOS has a hash table called a *device registry* which upon accesses to memory-mapped device registers to communicate with the appropriate (simulated) I/O device. For IRIX, the registers for each kind of device are mapped in specific ranges in the address space. For the porting of other OS, this 'registry' can be easily changed.

I/O device functionality for a console / network are provided by communicating with the real terminal / multiplexing with the real network on the host machine. The package *SimEther* uses the simulated network device can be used to copy files from the host to the simulated machine.

The simulated filesystem similarly resides as a (large) file on the host system; thus reads and writes to the simulated disk become reads / writes to this host file; simulated DMA transfers require copying of data from this file into the appropriate portion of the simulated memory. As the file may occupy several GB on the host, the simulated file system may be shared between different concurrent invocations of SimOS using a copy-on-write strategy (where modified blocks are stored separately for each application [9]).

Proper device simulation require an event to occur in future simulated time (eg. a timer interrupt). SimOS provides a *callback* facility in which it will invoke the required function at the required time. In multiprocess simulation, the callback mechanism is invoked only when switches occur, to reduce overheads.

The interfaces between the devices corresponds to that on an actual hardware, which allows a corresponding degree of modularity. For example, the CPU submits load/store requests to the memory system, which then has full control of the latency of satisfying that request. Similarly, with the disk device, a simple model can return a fixed latency (for situations where simulation speed is a priority, it can be made to satisfy the request immediately[5]) whereas a more complex model can also be used [9].

3.3.2 Event Collection and Classification

Full machine simulators can provide a vast amount of low-level event information; this must be filtered and translated to user-meaningful concepts [20]. For example, an PA-based data cache miss must be related to the corresponding process id and virtual address, and then mapped into specific application data structures.

SimOS thus provides means for the investigator supply the application-specific information required to perform these mappings. It separates the data management process into event generation and processing stages, allowing the investigator to customize (via a TcL script) the filtering, classification and reporting of events, without having to modify the simulator code itself [9]. From this classification, low-level events are transformed into higher-level information.

[5]This reduces in turn the amount of simulated time spent in the OS 'idle loop'.

SimOS provides a mechanism called *annotations* for user-definable execution monitoring. Annotations permit for example monitoring of entry into a given procedure, cache misses at a particular address, execution of particular opcodes, the raising of an exception, the reaching of a particular cycle count, etc. A high-level interface can be provided by ensuring the embedded TcL interpreter includes knowledge of the object file symbol table composition, and then by loading these tables of the target operating system / application. Thus an annotation can be entered eg.

```
annotation set pc kernel::vfault:START { incr vfaultCount }
```

which will be triggered when the PC enters the `vfault()` procedure in the kernel. The whole simulated machine state (registers and memory, architecture-specific) can be accessed in this way; the TcL interface is pre-defined. Using knowledge of the kernel's data structures, the current process id can for example be looked up. These corresponding TcL scripts are the main OS-dependent code in SimOS [20].

When an annotation-triggering event occurs, the associated script is interpreted. This is done by incorporating simple triggering code into each hardware model, eg. Mipsy would invoke `ExecuteAnnotations(TrapType, INTERRUPT);` each time a device interrupt occurs to invoker any associated trap-based annotations [9]. This procedure is optimized, eg. uses hashing techniques to quickly handle the common case, events that trigger no annotations. In the case of Embra, the annotation system is queried upon basic block translation to see if such a call for PC-related or instruction-type related events is required (evidently, any annotations triggered by data cache misses would always require such a call to be inserted). The cost of processing annotations is thus proportional to their number and complexity [9].

Annotation events can trigger checkpoints, switch simulator cores, as well as collect statistics. For the latter, it is also possible to classify events into different *selector buckets*, so that statistics can be sorted according to address ranges etc. Thus, common types of event counting classification can be made easy and efficient (unlike if directly implemented in annotations). The *bucket* consists of a specification the events that should be classified (eg. counts of instructions executed, TLB & cache misses); the *selector* is a specification of the execution phases of interest (eg. *system mode*: user, kernel and idle). Only one bucket selector is active at a particular time (this is controlled by a FSM implemented via annotations, eg. see Fig 5.3 [9]); all current hardware events are then funnelled into the active bucket.

Multiple selectors can co-exist. Accessing a counter within such a bucket generally involves a single level of indirection in the normal event counting code [20].

For finer-grained classification, eg. events based on individual instructions, the above mechanism would be unwieldy. *Address tables* are a special case of where a selector is based directly on the (code or data-related) address of the event, and thus could consist of many thousands of buckets. These are specified via scripts using the special `addressTable` commands. These can have varying granularity; hashing techniques are then used to determine the current bucket. Symbol table information can relate these address ranges back to the source code [9].

Higher-level events such as cache misses of various types (eg. col, invalidated, replaced) can be built up using *event filters*: like bucket selectors, a FSM can be used to build up extra knowledge on the occurrences of existing events, to trigger higher-level events (which may in turn trigger annotations)[6]. Such am event filter is often used in conjunction with an address table [9].

In general, *annotation layering* can create higher-level events. For example, process switch in/out events can be defined based on `pc`-based events; correspondingly, the process name and id can be maintained in TcL variables, which may be accessed by annotations that trigger on switch in/out events [9].

Buckets can also be arranged in a tree-like structure, called a *timing tree*, which is a system-wide tree with each process automatically added as a first level node. Scripts can then specify selectors based on (user specified parts of) the call tree within a process; a stack-like mechanism is used to determine the current bucket [9]. This enables for example the event collection the OS in terms of the various kernel services (eg. system calls, virtual memory fault processing, exceptions and interrupt handling). This mechanism isolated the activity of these services (even though they called common lower-level routines) and automatically handled difficult aspects such as nested interrupts and descheduled processes [9] (p92).

All of these mechanisms are encapsulated by SimOS in a library parameterized by an integer detail level:

[6] A possible alternative to the above example would be to build such information into the cache model themselves.

level	description	Mipsy overhead
0	no use of library	0
1	track process id	1%
2	1 + use system mode bucket selector	10%
3	2 + use system service timing trees	13%

These levels exhibit the usual speed-detail tradeoff: the overhead is roughly constant for any simulator cores; thus for Embra, levels 2 and 3 involve several hundred percent overhead [9].

The TcL interface is used also to configure SimOS (in a file `defaults.tcl`). The boot configuration is encapsulated in `init.simos`, which causes the kernel to boot in single-user mode.

3.3.3 Direct Execution in SimOS

While superceded by Embra in 1996, a *direct execution* core was initially used as the fast core to position workloads [21]. It can only be used when the host architecture is instruction set compatible to the target architecture (the OS would also have to be similar, handling signals and exceptions in a similar fashion). A further drawback is it only performs functional emulation, although at a factor of 2 or so slower than running on the host. As it involves some interesting techniques and ideas, these will be outlined here.

Here, the simulation (including that of the OS) runs as a user-level process (one for each CPU on the target architecture; these can thus operate in parallel on an SMP host) [21].

To simulate the trap architecture, the host OS notifies the SimOS process of exceptions via the UNIX *signals* mechanism. SimOS registers signal handlers to the OS that convert the signal information provided by the host OS into the form required by the target OS trap handlers, and then transfer control to them.

Privileged mode instructions, when executed at the user level, will raise an illegal instruction exception; thus they can be simulated in software using the signal handling mechanism mentioned above.

MMU simulation is performed by modelling the physical memory as a *memory-mapped* file (`mmap()`), where the host OS is used to map (where valid) each page into the (virtual) memory space of the SimOS process (see Figure 2 [21]). Thus the privileged instructions affecting the page tables are converted to host OS file mapping routines by the signals mechanism mentioned above. Accessing a VM address in the SimOS process that has no valid MMU mapping would correspond to writing to an unmapped file page in the host file pages mappings. This then results in an exception, which is later handled as page faults via the signal mechanism. Memory protection can be handled similarly.

A final issue is the target OS placement in virtual memory: this must be placed in the highest part of the memory of the user virtual address space on the host (see Figure 3 [21]).

Interrupts from devices are similarly simulated via signals. DMA accesses are implemented by the simulated devices accessing the physical memory file. A disk is simulated by a file which contains the information on the disk converted to raw disk format.

It remains to discuss how the simulator state becomes updated in direct execution mode. The SimOS process runs in its own single VM space; thus provided the simulated OS and the (currently running) application run in disjoint address spaces, direct execution will automatically update the simulate OS or application memory state without conflict. Normal registers probably cannot be saved. Thus one can only switch out of direct execution mode upon simulated instructions causing exceptions. The real host registers would then have to be added to the simulator state of Embra or Mipsy at that point. Conversely, when switching back to direct execution mode, host registers must be initialized by the simulator state.

When running more than one concurrent process, it is not clear how the problem of multiple address spaces can be solved for direct execution (unless this is avoided by modifying the simulated OS to ensure they are non-overlapping). Perhaps this limitation was the main reason for the dropping of this mode; the cited reason is that it is less amenable to cross-platform support than Embra [9].

The performance of direct execution suffers if there are a high proportion of privileged instructions need to be executed, as each of these requires a host interrupt and signal handling.

3.3.4 Embra in SimOS

Like Shade and SimICS, Embra uses dynamic translation as its central simulation mechanism. The unit of translation is a *basic block* of instructions, which like SimICS, are cached in a *translation cache* (TC); see Figure 3.1 of [26]. This is normally done in two passes; the first determines which registers are used and allocates host registers.

Thus, a simulation, again occurring in a single user-level host process, by iterating a *main dispatch loop*, which jumps to the translated blocks in the TC (performing the translation of not present in the TC).

To support full machine simulation, features including the trap architecture, MMU simulation (including multiple virtual address space), memory-access related exceptions, timer and I/O device interrupts, privileged instructions, memory-mapped I/O, and self-modifying code must be simulated and handled. In detail, this includes:

- the MMU is explicitly modelled, in turn requiring modelling of the TLB. As an MMU translation on every load and store instruction (indeed, on each instruction fetch), Embra uses several aggressive techniques to minimize this cost.

  To avoid the software cost of directly modelling a fully associative TLB, it uses a *MMU relocation array* (MMUra), indexed by virtual page number[7]; each element contains the corresponding physical address (if valid), TLB status and permissions. This is kept synchronized with the simulated TLB.

  Thus, a single lookup is required to perform the translation and check for exceptions (eg. TLB misses, page faults and segmentation faults).

  The explicit lookup is only made on the virtual address; a lookup on the currently executing process's *address space id* (ASID) is avoided (on a simulated TLB hit) as Embra adds MMUra entries only for the current ASID. Thus, on an MMU context switch, all MMUra entries marked as being in the TLB must be replaced.

  For instruction fetches, the translation need only occur upon translation of a block. However, instructions to detect TLB-related exceptions must be inserted into the translation of each basic block.

- kernel addressing modes such as uncached references for I/O devices are handled by putting an invalid entry in the MMUra, which is then detected and handled in the exception raising code.

- dynamically-generated (or modified) code are handled by flushing any code pages from the TC whenever such a page is written to.

Embra also has an in-built mode (dynamically selectable) that can model the memory behaviour of a multiprocessor with a single level of shared cache. This is sufficient to determine if a workload is bound by paging, I/O or memory bound. It is also useful for *positioning* a workload for a more accurate core (eg. Mipsy), and for determining whether the corresponding selected window of the simulation was representative of the benchmark [20].

When not in this mode, I/O devices are configured to perform pure emulation (instant request satisfaction, without simulated delay), to avoid wasting time simulating the OS idle loop [20].

There are 2 modes to simulate parallel execution:

1. (as for SimICS): replicate the the CPU state and MMUra for each simulated CPU; the simulation of these are then multiplexed (in 'time slices') within a single Embra process, thus sharing memory, devices etc.

   Shared code segments in the TC results in some performance savings.

   Here, for realism, the 'time slices' must be kept low (eg. 80 target cycles); thus it is important that the (context) switch overhead is small.

2. *parallel Embra*: simulate the CPU/MMU in multiple Embra processes with share the same simulated memory.

   This is quite fast; however, distortion will arise unless frequent barrier synchronizations are made[8]

The *chaining* of translated blocks involves replacing the jump back to the main dispatch loop by a jump to the translated following block. However, as different processes may be sharing the TC, physical addresses must be used in the TC (necessitating the overhead of translating the PC in the dispatch loop)[9]. Thus each (chained) translation is augmented with code checking that the PA of the PC is the same as that of the translation (via accessing MMUra[PC]). *Speculative chaining* is used Register indirect jumps (common in MIPS codes).

[7] This alone consumes ≈ 1 / page size of the host's virtual address space, assuming host and target have the same sized address space.

[8] It is not clear if this is working in the current implementation of Embra.

[9] As processes may share some code pages (at the same VA), but may have disjoint pages (also at the same VA), the chainings of spanning pages may not always be valid.

Annotations activated on memory address are simply activated by having a special invalid entry in the corresponding MMUra entry (this unfortunately would slow down all references to that page). In general, extra (user-customizable) statistics can be dynamically inserted simply by informing the Embra translator: this generally causes corresponding code to be inserted upon the next translation of the relevant instructions [20].

In cache mode, Embra can model a single level of a shared cache, apparently it must be direct-mapped. It uses a byte array indexed by the virtual cache line number, vQC, which as well as containing status in the cache, contains TLB & MMU relocation status (duplicated in the MMUra) in order to reduce checking overheads. Upon a cache miss, a memory system simulator is used to determine the stall. Note that the affected entries in the vQC must be updated upon simulated cache line or TLB replacement. Hence Embra's performance depends upon a high hit rate in the simulated workload. The vQC can be used optionally: otherwise a smaller physical memory array is used to model the cache [7].

Cache coherence is maintained by the memory system, which contains a bitmap for each line in physical memory, signifying which CPUs currently access the line. This enables quick invalidation of shared cache lines.

3.3.5 Mipsy and MXS in SimOS

Mipsy uses a traditional fetch-decode-execute loop to interpret each instruction [21]. In multiprocessor mode, Mipsy simulates each CPU an instruction at a time, thus attaining highly accurate interleavings. This also enables precise device interrupt timings and cache modelling.

It models a simple processor pipeline, charging a fixed latency for each instruction (without taking into accounts delays from dependencies and also, it seems, instruction groupings).

Some details of its implementation can be found in [9]; basically for each instruction cycle, there are procedure invokations corresponding to handling interrupts, reading memory (which in turn invokes a MMU emulation routine).

Mipsy uses a configurable model of a fixed latency (blocking) cache (2 levels). Each memory access passes through this model, and statistics can be collected. MXS can interface to a non-blocking model [9], which evidently requires a detailed CPU simulator to work.

It is surmised that in the case of Mipsy, the fixed delay is implemented simply by incrementing the simulated time counter.

Several memory systems are available, eg. a split-transaction, out-of-order completion memory bus (with multiple memory banks?), which in SMP mode, simulates snooping with an invalidation-based coherency protocol. Bus contention can be modelled. MXS can interface to FLASHlite, a cycle accurate implementation of the FLASH CC-NUMA memory architecture. Note that the simulation of DMA devices must also interact with the cache/memory model (eg. snooping effects).

In general, it is the easiest of the cores to understand and extend.

3.4 RSIM

RSIM [17, 18, 22] is a user-level detailed SPARC V9 SMP simulator. It corresponds to the SimOS MXS core for SPARC V9 in terms of the detail-performance tradeoff. However, it can only simulate user-level applications.

While in principle it can simulate un-modified binaries, for SMP simulation, special macros have to be placed in the source code at forks, joins and barriers in order to inform the simulator of important SMP events. In general, system calls are emulated.

RSIM is thus rather slow, but a speed-enhanced version called DirectRSIM [5] has recently been developed. The direct execution is as for SimOS (see Section 3.3.3), but is considerably simplified in the case user-level simulation. The application binary is instrumented [10] to perform timing information (determined from static analysis) for non-memory accesses instructions, and instrumented to invoke the memory analyser on the others.

The latter presents a problem for SMP loads: the value returned cannot in general be known until the timing information is complete. This problem is solved by having the timing simulator simulate the timings of all outstanding instructions since its last invocation.

The processes representing the different CPUs are thus required to synchronize only on memory transactions. It achieved an averaged speedup of 3.6 over RSIM with a loss of simulation accuracy of less than a few percent.

[10] This would require that at least the assembler source for the code to be available.

3.5 MPSAS

MPSAS is an SMP SPARC V8 simulator recently made available by Sun Laboratories [11]. It appears to perform a detailed hardware simulation, but it is not clear at this stage whether it is a full machine simulator in the sense and SimOS, and whether the level of detail it has is too great for simulation of realistic workloads (cites a simulation speed of thousands of cycles per second [11]).

It has a modular structure, like SimICS, and includes detailed SPARC device simulators. It is also easily configurable.

It appears that a V9 version is currently being developed.

3.6 Sulima

Sulima [27] is a framework for implementing full machine SMP simulators. It has been developed primarily by Patryk Zadarnowski of the Distributed Systems Group at UNSW.

Under Sulima, a Mipsy-like simulator for the 64 bit MIPS 4600 chip (Koala-4600) has been developed, originally for IRIX and porting is underway for the L4 microkernel. It also appears that cores for x86/FreeBSD and SPARC/Solaris have been implemented to some degree.

The Koala-4600 implementation is in C++; it heavily uses OO techniques to keep the code (relatively) modular and generic. It uses some buffering of instruction caches to speed up the instruction fetch, and issues blocks of instructions from each simulated CPU in MP mode. It currently models one level of cache.

An annotation interface (and gdb interface) still needs to be developed; Kaffe (JVM) or Guile (Scheme) are slated alternatives to TcL for its interface. At the time of writing, no documentation exists.

4 Issues in Evaluation of Candidate Tools as a basis to Theme 1

Note that the existing tool used by FL, called Shade, does not satisfy any requirement except 3.

A viable strategy for the Project is to extend a suitable existing simulation tool whose source code is available (candidates include SimICS[11], Shade, MPSAS, SimOS and RSIM) that can make these requirements easy to achieve.

Estimates from the authors of these tools of the amount of effort required to extend the tools are:

- SimICS: *are well beyond a small project* [13], for requirements 2–4 (requirements is 1 already satisfied); requirement 5 entails the port from SPARC V8 to V9.
- for SimOS: a full-time researcher already familiar with SimOS, SPARC V9 and Linux, *a year or more, realistically* [12], for requirements 5 only (requirement 1 is already satisfied).
- RSIM: *several months* [19], for requirement 1 (requirement 4 is already satisfied; requirement 5 is largely satisfied) [12].

In all of the above cases, requirement 3 is rather easy to satisfy, at least in a way that might not necessarily give good performance. Requirement 4 is also generally satisfied; however, the model may need to be modified for several specific target memory architectures.

The degree of support to be expected from this point on is unclear (probably small from SimOS and RSIM, as most of the researchers involved have moved on; probably none for Shade; for SimICS, it costs $10K pa).

There is a risk of a major problems/setbacks in such a task, especially where kernel-level simulation is concerned. Ensuring correctness and debugging of such a tool is also difficult.

Requirement 1 is apparently the most difficult to achieve. Only SimICS and SimOS support this (requirement 6 also); these and Shade support requirement 7.

5 Conclusions

SimICS would be the best choice if its source codes were available and its licensing costs (esp. for FL to use) were acceptable. Not only is it closest to satisfying all requirements, its modular, portable and relatively clean

[11] It is not clear that the source code for the simulator core, needed for requirement 5 and possibly 2, would ever be made available to such a project

[12] It is not clear that the researcher giving this estimate full understands the full implications of requirement 1, but also pointed out that extension for partial system level simulation has recently been carried out at the University of Utah [19].

design would make code reading and modification relatively easy. However, its host memory-efficient STC implementation could be incorporated to reduce the (rather onerous) virtual memory requirements of Embra, and/or to improve the efficiency of Mipsy.

At present, the port of SimOS to SPARC V9 seems the most promising course. However, a first examination of the (MIPS) source code reveals that it is very hard to read, with little supporting (internal or external) documentation. This is particularly the case for Embra (note that the MXS core would not need to be ported to satisfy the requirements). It is not obvious which parts are machine or OS dependent. On the other hand, comparing code for the Alpha port with that of the MIPS may help in this way.

A factor strongly in favour of SimOS is its sophisticated data collection mechanisms; furthermore, it has been used for similar purposes (OS development, compiler applications, database workloads and memory architecture studies) as are intended for the CAP Project Phase III.

As extending RSIM for requirements 1 and 7 seems the least straightforward, this does not seem the best approach. A further problem is that satisfactorily meeting requirement 6 may be very difficult (or impossible, even with DirectRSIM). Still, sections of the source code may be useful in porting SimOS, so these are definitely worth studying.

The viability of using MPSAS as a basis is yet to be determined at the time of writing. While it seems unlikely that it will be fast enough for requirement 7, some code modules for devices etc might prove useful.

At the time of writing, as the details (and source) of Sulima are still unavailable, it is difficult to evaluate whether it is a viable choice. Potentially, it could be the easiest way to get a running SPARC V9 simulator; however, it would then require implementing the data collection and debugging infrastructure, which may be quite a large task.

References

[1] Pradip Bose and Thomas M. Conbte. Performance Analysis and its Impact on Design. *IEEE Computer*, pages 41–49, May 1998.

[2] Cristina Cifuentes and Mike Van Emmerik. UQBT: Adaptible Binary Translation at Low Cost. *IEEE Computer*, pages 60–66, March 2000.

[3] Robert F. Cmelik and David Keppel. Shade: A Fast Instruction-Set Simulator for Execution Profiling. Technical Report SMLI 93-12, Sun Microsystems Laboratories Inc, 1993.

[4] Robert F. Cmelik and David Keppel. Shade: A Fast Instruction-Set Simulator for Execution Profiling. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 128–137, May 1994.

[5] Murthy Durbhakula, Vijay S. Pai, and Sarita Adve. Improving the Speed vs. Accuracy Tradeoff for Simulating Shared-Memory Multiprocessors with ILP Processors. In *Proceedings of the Fifth International Symposium on High Performance Computer Architecture*, pages 23–32, January 1999.

[6] Susan J. Eggers, David Keppel, Eric J. Koldinger, and Henry M. Levy. Techniques for Efficient Inline Tracing on a Shared-Memory Multiprocessor. In *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 37–47, May 1990.

[7] Steve Herrod et al. The SimOS Simulation Environment. http://simos.stanford.edu, October 1997.

[8] R. Giorgi, C.A. Prete, G. Primac, and L. Riccardi. Trace Factory: Generating Workloads for Trace-Driven Simulation of Shared-bus Multiprocessors. *IEEE Concurrency*, pages 54–67, Fall 1997.

[9] Stephen A. Herrod. *Using Complete Machine Simulation to Understand Computer System Behavior.* PhD thesis, Stanford University, February 1998. viii+120p.

[10] M. Kawaba. MUltiprocessor system Simulator using Cache Access Traces. Private comunications, April 1999.

[11] Sun MicroSsytems Laboratories. The Multiprocessor SPARC Architecuture Simulator (MPSAS) User Guide. Technical report, Sun Microsystems Inc, July 1999.

[12] Bob Lantz. Private comunications, January 2000.

[13] P. Magnusson. Private comunications, January 2000.

[14] Peter S. Magnusson, Fredrik Dahlgren, Hekan Grahn, Magnus Karlsson, Fredrik Larsson, Fredrik Lundholm, Andreas Moestedt, Jim Nilsson, Per Stenstrvm, and Bengt Werner. SimICS/sun4m: A Virtual Workstation. In *Usenix Annual Technical Conference*, June 1998.

[15] Peter S. Magnusson and Bengt Werner. Some Efficient Techniques for Simulating Memory. In *28th Annual Simulation Symposium*, 1995.

[16] T. Ozawa. Discussions on the Phase III Themes. Private comunications, 1999–2000.

[17] Vijay S. Pai, Parthasarathy Ranganathan, and Sarita V. Adve. RSIM: An Execution-Driven Simulator for ILP-Based Shared-Memory Multiprocessors and Uniprocessors. In *Proceedings of the Third Workshop on Computer Architecture Education*, February 1997. Also appears in IEEE TCCA Newsletter, October 1997.

[18] Vijay S. Pai, Parthasarathy Ranganathan, and Sarita V. Adve. RSIM Reference Manual. version 1.0. Technical Report 9705, Electrical and Computer Engineering Department, Rice University, July 1997.

[19] P. Rangathan. Private comunications, February 2000.

[20] Mendel Rosenblum, Edouard Bugnion, Scott Devine, and Steve Herrod. Using the SimOS Machine Simulator to Study Complex Computer Systems. *ACM TOMACS Special Issue on Computer Simulation*, 1997.

[21] Mendel Rosenblum, Stephen A. Herrod, Emmett Witchel, and Anoop Gupta. Complete Computer System Simulation: The SimOS Approach. *IEEE Parallel and Distributed Technology*, Fall 1995.

[22] Dan Sorin, Vijay S. Pai, Sarita V. Adve, Mary K. Vernon, and David Wood. Analytic Evaluation of Shared-Memory Systems with ILP PProcessors. In *Proceedings of the 25th International Symposium on Computer Architecture*, June 1998.

[23] Craig B. Stunkel, Bob Janssens, and W. Kent Fuchs. Address Tracing for Parallel Machines. *IEEE Computer*, 24(1):31–38, January 1999.

[24] Jack E. Veenstra. MINT Tutorial and User Manual. Technical Report 452, University of Rochester Computer Science Department, May 1993.

[25] Bengt Werner and Peter S. Magnusson. A Hybrid Simulation Technique for Enabling Performance Characterization of Large Software Systems. In *Mascots 97*, pages 73–80, 1997.

[26] Emmett Witchel and Mendel Rosenblum. Embra: Fast and Flexible Machine Simulation. In *Proceedings of ACM SIGMETRICS '96: Conference on Measurement and Modeling of Computer Systems*, Philadelphia, 1996.

[27] Patryk Zadarnowski. The Sulima simulator. http://www.sulima.org, July 2000.